

# Lesson 10

## Algorithm Design:

**Wholeness of the lesson:** Algorithm Design is an intelligent approach to solving problems with algorithms. Rather than simply trying to tackle a problem haphazardly, one can determine whether the problem has the characteristics that make it easy to solve using one of many known algorithm design strategies.

**Maharishi's Science of Consciousness:** The textbook of SCI, the Bhagavad Gita, declares "Yogastah Kuru Karmani" – Established in Being, perform action. When awareness has a chance to be bathed in the field of pure orderliness, activity afterwards has an orderly quality that naturally leads to success and achievement.

# Three major techniques:

- ◆ Divide-and-Conquer
- ◆ Dynamic Programming
- ◆ The Greedy Method

# Applications of Each Technique

## ◆ Divide-and-Conquer

- Binary Search (and some operations on a BST)
- MergeSort
- QuickSort

## ◆ Dynamic Programming

- Revised Recursive Fibonacci
- Edit Distance
- SubsetSum
- Knapsack

## ◆ The Greedy Method

- Shortest Path (in a graph - later)
- Minimum Spanning Tree (in a graph - later)

# Divide and Conquer

Involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then “merging” or “marrying” the solutions to the subproblem(s) to produce a solution to the original problem.

# Divide and Conquer Strategy

The method:

- **Divide** the problem into subproblems.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems into a solution to the problem.

# Binary Search

**Algorithm** search(A,x)

*Input:* An already sorted array A with n elements and search value x

*Output:* true or false

**return** binSearch(A, x, 0, A.length-1)

**Algorithm** binSearch(A, x, lower, upper)

*Input:* Already sorted array A of size n, value x to be searched for in array section A[lower]..A[upper]

*Output:* true or false

**if** lower > upper **then return** false

mid  $\leftarrow$  (upper + lower)/2

**if** x = A[mid] **then return** true

**if** x < A[mid] **then**

**return** binSearch(A, x, lower, mid - 1)

**else**

**return** binSearch(A, x, mid + 1, upper)

# MergeSort

**Algorithm** *mergeSort*( $S$ )

**Input** sequence  $S$  with  $n$

**Output** sequence  $S$  sorted

**if**  $S.size() > 1$  **then**

$(S_1, S_2) \leftarrow partition(S, n/2)$

*mergeSort*( $S_1$ )

*mergeSort*( $S_2$ )

$S \leftarrow merge(S_1, S_2)$

**return**  $S$

# Divide and Conquer Doesn't Always Work Efficiently

- ❖ For Divide and Conquer to be effective, it must be possible to break up the original problem into *non-overlapping* subproblems. (Overlapping subproblems: the recursion tends to solve the same subproblems over and over.)
  - ❑ Example: In MergeSort, the steps of recursive sorting of the left half of the list do not affect, and are not affected by, the steps of the sorting of the right half of the list
- ❖ If something similar to Divide and Conquer is attempted when problems are overlapping, it may result in many redundant computations.
  - ❑ Example: Recursive Fibonacci

**Algorithm** fib(n)

**Input:** a natural number n

**Output:** F(n)

**if** (n = 0 || n = 1) **then return** n

**return** fib(n-1) + fib(n-2)



# Dynamic Programming

- ◆ *Dynamic programming* is a technique that has been used to find more efficient solutions to NP-hard (more on this later) problems, though often even these solutions are still exponential.
- ◆ The idea: Sometimes problems can be broken down into *overlapping* subproblems, which can be solved, and whose solutions can be combined in some way to obtain a solution to the main problem. Solutions to subproblems are stored and combined stage by stage to produce a solution to the main problem.

# (continued)

- ◆ When such a problem exhibits the following characteristics, it can in many cases be tackled using dynamic programming:
  - *Overlapping subproblems* – the subproblems “overlap” – the recursion tends to solve the same subproblems over and over (example: recursive fibonacci)
  - *Optimal substructure* – an optimal solution is composed of a combination of optimal solutions to subproblems

# Dynamic Programming Example: Fibonacci

- ◆ To generate the  $n$ th Fibonacci number, the subproblems are computation of the  $k$ th Fibonacci numbers for  $k < n$ .
- ◆ To prevent redundant computation, solutions to subproblems can be stored in a table and accessed whenever needed during execution of the algorithm

# Dynamic Programming Solution to Recursive Fibonacci

- ❖ Demo: `lecture_10.fastfib`
- ❖ In this dynamic programming solution, the integer table stores computations as they are made.
- ❖ Computations are made only after consulting the table to see if a computed value is already available.

# Dynamic Programming: The Edit Distance Problem

◆ The Problem: Given two strings, what is the smallest number of transformations (delete, insert, substitute) to transform one to the other?

- Example: Transform "duck" to "tug"
  - ◆ First try: (1) delete d-u-c-k (2) insert t-u-g  
Total # transformations = 7
  - ◆ Second try: duck -> tuck -> tugk -> tug  
Total # transformations = 3

◆ Applications:

- Approximate String Matching
- Spell checking
- Google – finding similar word variations
- DNA sequence comparison

# Recursive Solution

- ◆ When transforming one word to another one, consider the last characters of strings  $s$  and  $t$ .
- ◆ If we are lucky enough that they ALREADY match,
  - then we can simply recursively find the edit distance between the two strings left when we delete this last character from both strings.
- ◆ Otherwise, we MUST make one of three changes then recursively compute the edit distance:
  - 1) delete the last character of string  $s$
  - 2) delete the last character of string  $t$
  - 3) change the last character of string  $s$  to the last character of string  $t$ .

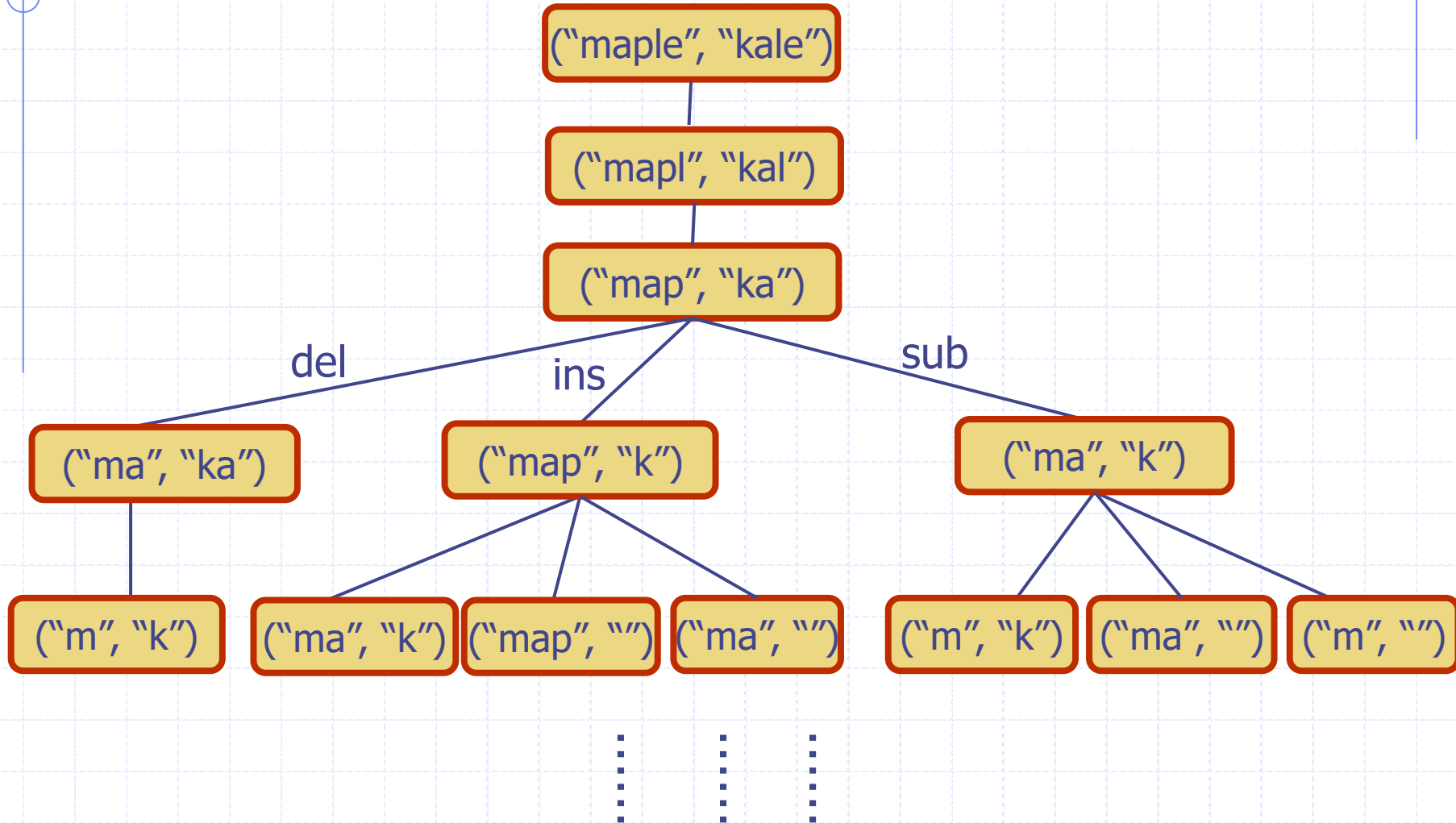
Note: In our recursive solution, we must note that the edit distance between the empty string and another string is the length of the second string. (This corresponds to having to insert each letter for the transformation.)

# Recursive Solution

◆ An outline of our recursive solution is as follows:

- 1) If either string is empty, return the length of the other string.
- 2) If the last characters of both strings match, recursively find the edit distance between each of the strings without that last character.
- 3) If they don't match then return  $1 +$  minimum value of the following three choices:
  - a) Recursive call with the string  $s$  w/o its last character and the string  $t$
  - b) Recursive call with the string  $s$  and the string  $t$  w/o its last character
  - c) Recursive call with the string  $s$  w/o its last character and the string  $t$  w/o its last character.

# Execution Example





# Recursive Solution

**Algorithm** recursiveED( $A_i, B_j$ )

**Input:** two strings  $A_i = a_1 \dots a_i$  and  $B_j = b_1 \dots b_j$

**Output:** the edit distance for  $A_i$  and  $B_j$

**if**  $i = 0$  **then**

**return**  $j$

**if**  $j = 0$  **then**

**return**  $i$

**if**  $A[i] = B[j]$  **then**

**return** recursiveED( $A_{i-1}, B_{j-1}$ )

**else**

**return**  $1 + \min(\text{recursiveED}(A_{i-1}, B_j),$   
        recursiveED( $A_i, B_{j-1}$ ),  
        recursiveED( $A_{i-1}, B_{j-1}$ ))

# Running time

Let  $n$  be the sum of the lengths of the two strings.

$$T(0) = a$$

$$T(1) = c$$

$$T(n) = T(n-1) + T(n-1) + T(n-2) + d \quad (\text{for some } c, d > 0)$$

By an exercise in Lab 2, we conclude that

$$T(n) \text{ is in } \Omega((\sqrt{3})^n)$$

In other words,  $T(n)$  is exponential or worse.

# The Edit Distance Problem

- ◆ Now, how do we use this to create a DP solution?
  - Let us consider the subproblems. What are the subproblems?
  - Are they overlapping?
  - If so, how can we store the solutions to overlapping subproblems?

# The Edit Distance Problem

◆ Now, how do we use this to create a DP solution?

- Let us consider the subproblems. What are the subproblems?

**Answer:** solving the edit distance problem for prefixes of s and t.

- Are they overlapping?

**Answer:** Yes

- If so, how can we store the solutions to overlapping subproblems?

**Answer:** All the possible recursive calls we are interested in are determining the edit distance between prefixes of s and t. We simply need to store the answers to all the possible recursive calls in a two dimensional array.

# Dynamic Programming Solution

◆ Let  $A = a_1 \dots a_i \dots a_n$  and  $B = b_1 \dots b_j \dots b_m$

define  $A_i = a_1 \dots a_i$  and  $B_j = b_1 \dots b_j$

$$D[i][j] = D_{i,j} = \text{EditDistance}(A_i, B_j)$$

that is,  $D_{i,j}$  is the edit distance for the prefixes  $A_i$  and  $B_j$

Note: each recursive call gives us  $D_{i,j}$

# Recursive Dynamic Programming Solution

**Algorithm** recursiveED( $A_i, B_j$ )

on first call, initialize a two dimensional array  $D[i+1][j+1]$  with all cells value -1

**if**  $i = 0$  **then return**  $j$

**if**  $j = 0$  **then return**  $i$

**if**  $D[i][j] \neq -1$  **then return**  $D[i][j]$

**if**  $A[i] = B[j]$  **then**

**if**  $D[i-1][j-1] = -1$  **then**

$D[i-1][j-1] \leftarrow \text{recursiveED}(A_{i-1}, B_{j-1})$

$D[i][j] \leftarrow D[i-1][j-1]$

**else**

**if**  $D[i-1][j-1] = -1$  **then**

$D[i-1][j-1] \leftarrow \text{recursiveED}(A_{i-1}, B_{j-1})$

**if**  $D[i-1][j] = -1$  **then**

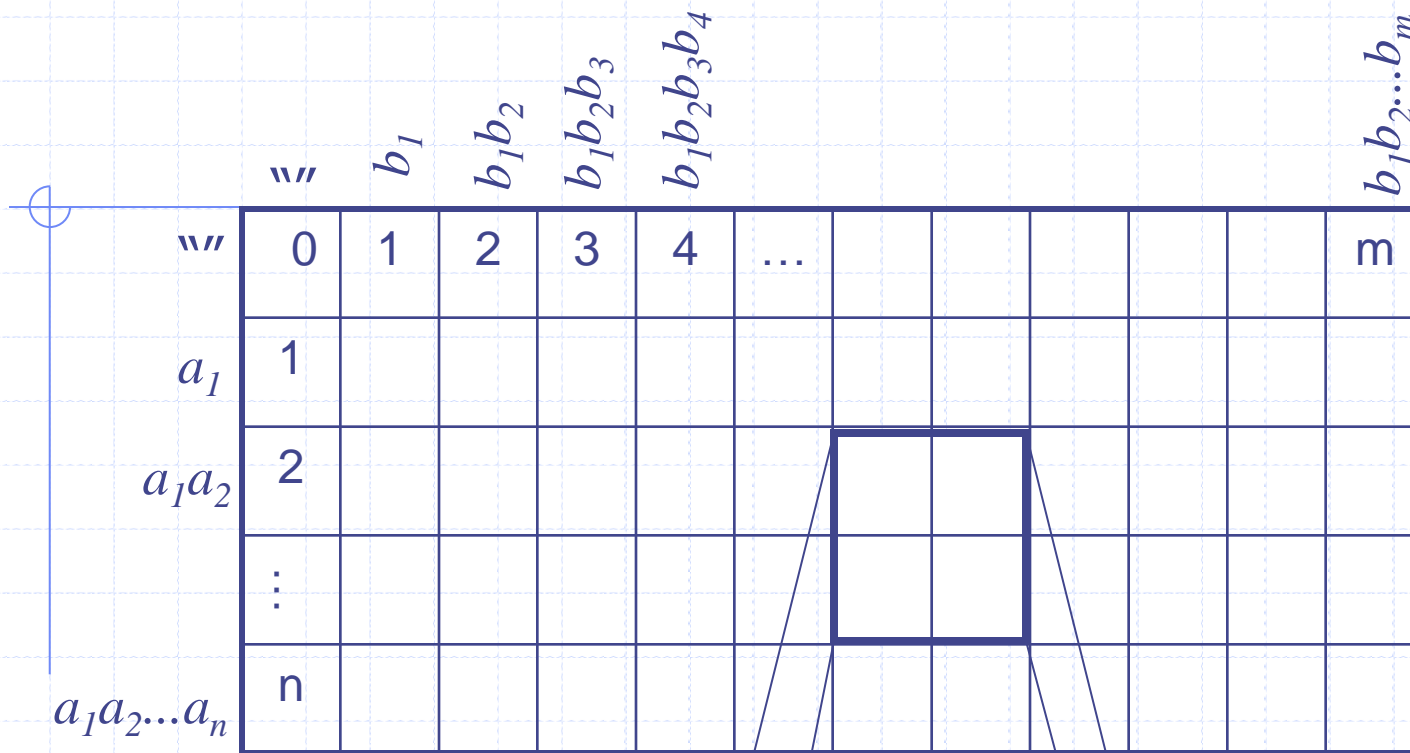
$D[i-1][j] \leftarrow \text{recursiveED}(A_{i-1}, B_j)$

**if**  $D[i][j-1] = -1$  **then**

$D[i][j-1] \leftarrow \text{recursiveED}(A_i, B_{j-1})$

$D[i][j] \leftarrow 1 + \min(D[i-1][j-1], D[i-1][j], D[i][j-1])$

**return**  $D[i][j]$



The diagram shows a large grid representing a dynamic programming table. The columns are indexed by  $b$  values:  $b_1, b_1b_2, b_1b_2b_3, b_1b_2b_3b_4, \dots, b_1b_2\dots b_m$ . The rows are indexed by  $a$  values:  $a_1, a_1a_2, \dots, a_1a_2\dots a_n$ . A 2x2 subgrid is highlighted in the center, with arrows pointing to a callout box below. The callout box shows the four subproblems that lead to the highlighted cell:  $D_{i-1,j-1}$  (top-left),  $D_{i-1,j}$  (top-right),  $D_{i,j-1}$  (bottom-left), and  $D_{i,j}$  (bottom-right).

	"	$b_1$	$b_1b_2$	$b_1b_2b_3$	$b_1b_2b_3b_4$	...						$b_1b_2\dots b_m$
"	0	1	2	3	4	...						m
$a_1$	1											
$a_1a_2$	2											
$\vdots$												
$a_1a_2\dots a_n$	n											

Iterative Dynamic  
Programming Solution

$D_{i-1,j-1}$

$D_{i-1,j}$

$D_{i,j-1}$

$D_{i,j}$

# Iterative Dynamic Programming Solution

- ◆ Let  $A = a_1 \dots a_i \dots a_n$  and  $B = b_1 \dots b_j \dots b_m$   
define  $A_i = a_1 \dots a_i$  and  $B_j = b_1 \dots b_j$   
 $D[i][j] = D_{i,j} = \text{EditDistance}(A_i, B_j)$   
that is,  $D_{i,j}$  is the edit distance for the prefixes  $A_i$  and  $B_j$

- ◆ **Base cases:**

$$D_{0,0} = 0$$

$$D_{i,0} = i \text{ for } 1 \leq i \leq n$$

$$D_{0,j} = j \text{ for } 1 \leq j \leq m$$

The edit distance between an empty string and another string  $s$  is the length of  $s$ .



# Iterative Dynamic Programming Solution

- ◆ Based on the observation from recursive solution, we come up with the following **Recurrence equation**:

if ( $A[i] \neq B[j]$ )

$$D_{i,j} = \min \left\{ \begin{array}{l} D_{i-1,j-1} + 1, \\ D_{i-1,j} + 1, \\ D_{i,j-1} + 1 \end{array} \right\}$$

if ( $A[i] == B[j]$ )

$$D_{i,j} = D_{i-1,j-1}$$

# Iterative Dynamic Programming Solution – building table

- ❖ Example: compute edit distance between "DUCK" and "TUG"
- ❖ Filling out first row and first column for the table.

	""	"T"	"TU"	"TUG"
""	0	1	2	3
"D"	1			
"DU"	2			
"DUC"	3			
"DUCK"	4			

# Iterative Dynamic Programming Solution – building table

- ❖ Example: compute edit distance between “DUCK” and “TUG”
- ❖ Filling out the rest cells for the table ...

	“”	“T”	“TU”	“TUG”
“”	0	1	2	3
“D”	1	1	2	3
“DU”	2	2	1	2
“DUC”	3	3	2	2
“DUCK”	4	4	3	3

# Iterative Dynamic Programming Solution

**Algorithm** EditDistance(A,B)

**Input:** two strings  $A = a_1 \dots a_n$  and  $B = b_1 \dots b_m$

**Output:** the edit distance for A and B

initiate a two dimensional array  $D[n+1][m+1]$

$D[0][0] = 0$

**for**  $i \leftarrow 1$  **to**  $n$  **do**  $D[i][0] \leftarrow i$

**for**  $j \leftarrow 1$  **to**  $m$  **do**  $D[0][j] \leftarrow j$

**for**  $i \leftarrow 1$  **to**  $n$  **do**

**for**  $j \leftarrow 1$  **to**  $m$  **do**

**if**  $A[i] = B[j]$  **then**

$D[i][j] \leftarrow D[i-1][j-1]$

**else**

$D[i][j] \leftarrow \min( D[i-1][j] + 1,$   
                                     $D[i][j-1] + 1,$   
                                     $D[i-1][j-1] + 1)$

**return**  $D[n][m]$

# Running time

- ◆  $O(mn)$  apparently. ( $m, n$  are the lengths of the strings.)
- ◆ This algorithm is discovered by Wagner Fischer in the 1970s.

# Summary

## Five Steps of Dynamic Programming:

- ◆ 1. Identify subproblems. (and subproblems must be overlapping in order to use dynamic programming technique)
- ◆ 2. Characterize the structure of a solution by recursively define the solution in terms of solutions to subproblems
- ◆ 3. Locate subproblem overlap
- ◆ 4. Store overlapping subproblem solutions for later retrieval
- ◆ 5. Construct an optimal solution from the computed information gathered during steps 3 and 4

# Memoization

- ◆ The basic idea
  - Design the natural recursive algorithm
  - If recursive calls with the same arguments are repeatedly made, then memoize the inefficient recursive algorithm
    - ◆ Save these subproblem solutions in a table so they do not have to be recomputed
- ◆ Implementation
  - A table is maintained with subproblem solutions and the control structure for filling in the table occurs during normal execution of the recursive algorithm
  - Often we can transform it into an iterative solution (which also called bottom-up solution.)

# Main Point

Dynamic Programming is an algorithm design technique that arrives at an optimal solution by computing optimal solutions to overlapping subproblems, storing the results (memoization) to avoid redundant computations, and then combining subproblem solutions to obtain the final solution. In SCI, it is observed that to restore completeness in the life of the individual – to solve the problem of life as a human being – we must restore the *memory* of our unbounded nature. For that, we repeatedly open awareness to its unbounded nature, through the process of transcending.



# Dynamic Programming: The Subset Sum Problem

The Subset Sum optimization problem says: We have set  $S = \{s_0, s_1, \dots, s_{n-1}\}$  of  $n$  positive integers and a non-negative integer  $k$ . Find a subset  $T$  of  $S$  so that the sum of the  $s_r$  in  $T$  is  $k$ .

$$\sum_{s_r \in T} s_r = k.$$

# SubsetSum: Recursive Solution

A recursive solution is based on the following observation:

We are seeking a  $T \subseteq S = \{s_0, s_1, \dots, s_{n-2}, s_{n-1}\}$  whose sum is  $k$ . Such a  $T$  can be found if and only if one of the following is true:

- (1) A subset  $T_1$  of  $\{s_0, s_1, \dots, s_{n-2}\}$  can be found whose sum is  $k$ , OR
- (2) A subset  $T_2$  of  $\{s_0, s_1, \dots, s_{n-2}\}$  can be found whose sum is  $k - s_{n-1}$

If (1) holds, then the desired set  $T$  is  $T_1$ . If (2) holds, the desired set  $T$  is  $T_2 \cup \{s_{n-1}\}$ .

The recursion proceeds by considering progressively smaller subsetsum problems, as the input set evolves from  $\{s_0, \dots, s_{n-1}\}$  to  $\{s_0, \dots, s_{n-2}\}$  to  $\{s_0, \dots, s_{n-3}\}$  to ... to  $\{s_0\}$ .

## (continued)

A recursive formula for computing  $T \subseteq \{s_0, s_1, \dots, s_{n-1}\}$  whose sum is  $k$  is:

$$T = \begin{cases} T_1 & \text{where } T_1 \subseteq \{s_0, \dots, s_{n-2}\} \text{ and } \sum T_1 = k \\ T_2 \cup \{s_{n-1}\} & \text{where } T_2 \subseteq \{s_0, \dots, s_{n-2}\} \text{ and } \sum T_2 = k - s_{n-1} \\ \text{NULL} & \text{otherwise} \end{cases}$$

The base case for the recursion is the case in which the input set has been reduced just to  $\{s_0\}$ . The possible values for a solution  $T \subseteq \{s_0\}$  are given by:

$$T = \begin{cases} \{\} & \text{if } k = 0 \\ \{s_0\} & \text{if } k = s_0 \\ \text{NULL} & \text{otherwise} \end{cases}$$

The base case tells us that unless self-calls eventually arrive at a subsetsum problem with input set  $\{s_0\}$  and whose target value is either 0 or  $s_0$ , then there is no solution to the problem.

# (continued)

Algorithm *RecSubsetSum*( $S, k$ )

Input:  $S = \{s_0, s_1, \dots, s_{n-1}\}$  positive integers,  
       $k$  nonnegative integer

Output:  $T \subseteq S$  for which  $\text{sum}(T) = k$

//base case

if  $S.\text{size}() = 1$  then

    if  $k = 0$  then return  $\{\}$

    else if  $k = s_0$  then return  $\{s_0\}$

    else return *NULL*

$(S, \text{last}) \leftarrow S.\text{removeLast}()$

$T \leftarrow \text{RecSubsetSum}(S, k)$

if  $T$  not *NULL* then

    return  $T$

$T \leftarrow \text{RecSubsetSum}(S, k - \text{last})$

if  $T$  not *NULL* then

    return  $T \cup \{\text{last}\}$

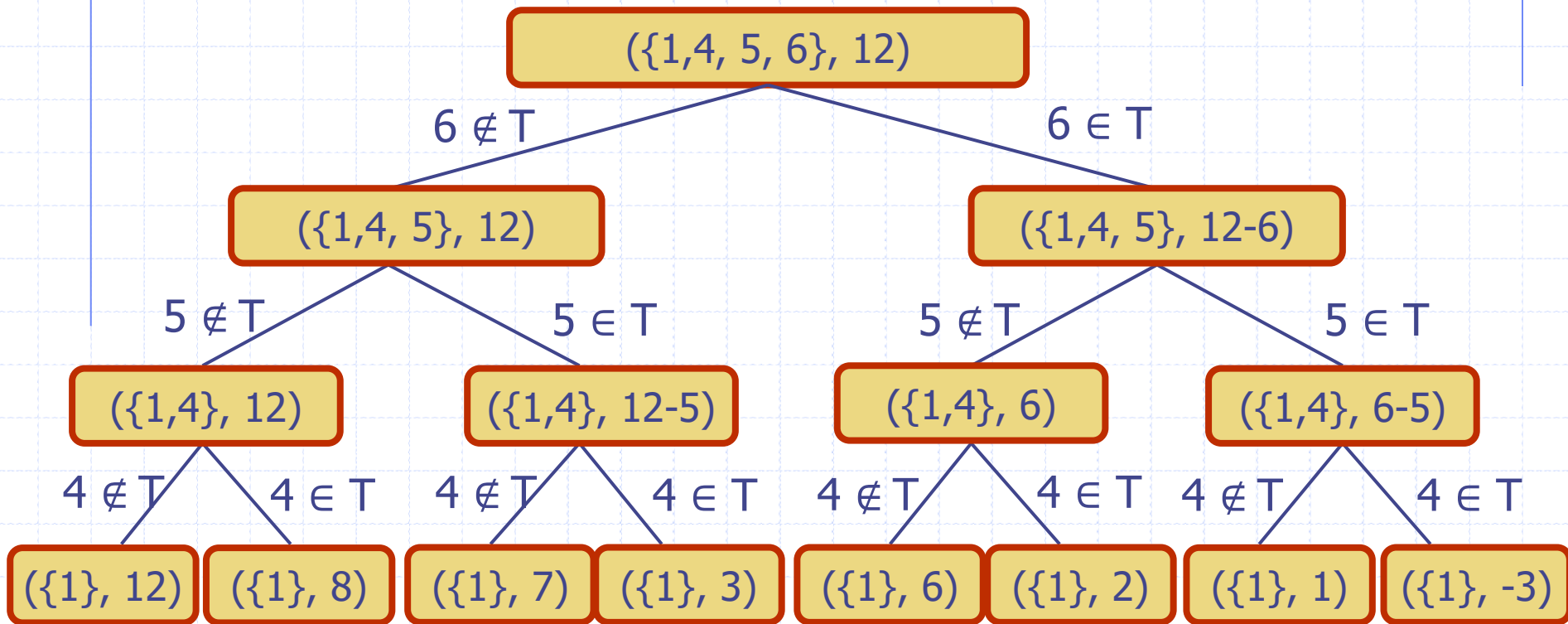
return *NULL*

- The recursive algorithm tries to find a solution  $T$  for  $(\{s_0, s_1, \dots, s_{n-2}, s_{n-1}\}, k)$  by checking if a solution exists for either of the subproblems  
     $(\{s_0, s_1, \dots, s_{n-2}\}, k)$   
     $(\{s_0, s_1, \dots, s_{n-2}\}, k - s_{n-1})$
- To find these, it seeks solutions to smaller subproblems.
- Note that when a self-call, running on some  $(S, k)$ , returns a value for  $T$ , if  $T$  is not *NULL*,  $T$  is guaranteed to have  $\text{sum} = k$ .

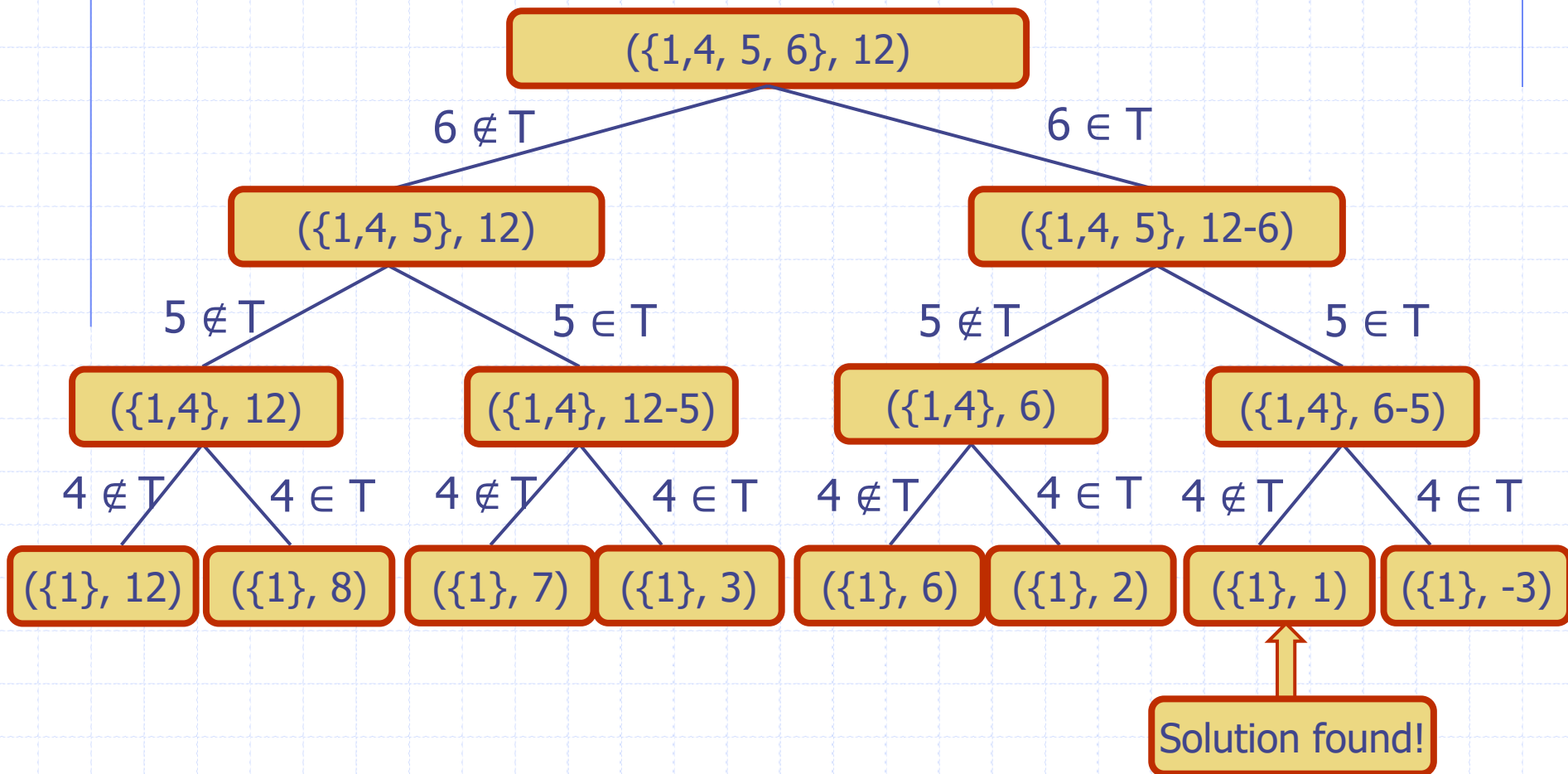
# SubsetSum Recursion Tree

- ◆ An execution of SubsetSum may be depicted by a binary tree
  - each node represents a recursive call of SubsetSum and stores the set  $S$  and  $k$
  - the link between two nodes tests whether the last element is in the solution.
  - the root is the initial call
  - the leaves are calls on set of size 1

# Execution Example



# Execution Example (cont.)



# Running time

- ◆ We use the technique of counting self-calls to compute the running time.
- ◆ # of self-calls is approximately equal to # of nodes in the recursion tree. The recursion tree is a completely filled binary tree, thus having  $2^{h+1}-1 = 2^n-1$  nodes, where  $h$  = height of tree,  $n$  = size of  $S$  (and  $h=n-1$ )
- ◆ So the running time of recursive SubsetSum is  $\Theta(2^n)$ .
- ◆ For larger input sets  $S$ , the recursive solution will repeatedly recalculate solutions for the smaller subproblems (recall how this happened with recursive Fibonacci). See Demo

`lecture_10.subsetsum.RecursiveSS`



# DP Solution: Improve Performance by Storing Computations

//Initialize hashtable H

Algorithm *RecSubsetSum*( $S, k$ )

Input:  $S = \{s_0, s_1, \dots, s_{n-1}\}$  positive integers,  
 $k$  nonnegative integer

Output:  $T \subseteq S$  for which  $\text{sum}(T) = k$

//base case

if  $S.\text{size}() = 1$  then

    if  $k = 0$  then return  $\{\}$

    else if  $k = s_0$  then return  $\{s_0\}$

    else return NULL

$(S, \text{last}) \leftarrow S.\text{removeLast}()$

//read stored computation if possible

$T \leftarrow H.\text{get}((S, k))$

if  $T$  not found then

$T \leftarrow \text{RecSubsetSum}(S, k)$

$H.\text{put}((S, k), T)$  //store for future use

if  $T$  not NULL then

    return  $T$

//read stored computation if possible

$T \leftarrow H.\text{get}((S, k - \text{last}))$

if  $T$  not found

$T \leftarrow \text{RecSubsetSum}(S, k - \text{last})$

$H.\text{put}((S, k - \text{last}), T)$  //store for future use

if  $T$  not NULL then

    return  $T \cup \{\text{last}\}$

return NULL

- In this version, the hashtable  $H$  is consulted before performing another self-call. If the computation has already been made and stored, the stored value is used.
- Performance using memoization in this way is greatly enhanced; all redundant computations are eliminated. See demo

lecture\_10.subsetsum.DynamicProgRecursive

# Running Time of DP Solution: Organizing Stored Computations in a Table

- ◆ We can organize the stored computations in the recursive algorithm in a table.

$A_{i,j}$	0	1	...	j	...	k-1	k
0							
1							
...							
i				$A_{i,j}$			
...							
n-2							
n-1							$A_{n-1,k}$

- ◆  $A_{i,j}$  is a solution to the SubsetSum problem  $(\{s_0, s_1, s_2, \dots, s_i\}, j)$

# Running Time of DP Solution: Organizing Stored Computations in a Table

- ◆ We can organize the stored computations in the recursive algorithm in a table.

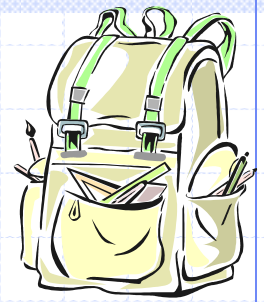
$A_{i,j}$	0	1	...	j	...	k-1	k
0							
1							
...							
i				$A_{i,j}$			
...							
n-2							
n-1							$A_{n-1,k}$

- ◆ Note that each self call tries to compute a cell in this table. (Same cell will not be computed more than once since solutions are stored.) Running time is  $O(\# \text{ of self calls}) = O(\# \text{ of cells in the table}) = O(kn)$

# Pseudo-polynomial time

- The dynamic programming solution to SubsetSum runs in  $O(kn)$
- However,  $k$  may be much bigger than  $n$ , and even if  $k$  is  $\Theta(n)$ , the true running time is based on the number of bits in  $k$ , not on the value of  $k$ . So even this algorithm runs in exponential time in terms of input size.
- Whenever an algorithm's running time is polynomial in the numeric value of the input, but is exponential in the size of the input (size = the number of bits required to represent it), the algorithm is said to run in pseudo-polynomial time.

# Dynamic Programming: Knapsack Problem



**The Problem.** Given a set  $S = \{s_0, s_1, \dots, s_{n-1}\}$  of items, weights  $\{w_0, w_1, \dots, w_{n-1}\}$  and values  $\{v_0, v_1, \dots, v_{n-1}\}$  and a max weight  $W$ , find a subset  $T$  of  $S$  whose total value is maximal subject to constraint that total weight is at most  $W$ .

**Observation.** If  $T$  is a solution, either  $s_{n-1}$  belongs to  $T$  or it does not.

1. If it does not, then  $T$  is a solution to the Knapsack problem with items  $\{s_0, s_1, \dots, s_{n-2}\}$  weights  $\{w_0, w_1, \dots, w_{n-2}\}$  values  $\{v_0, v_1, \dots, v_{n-2}\}$  and max weight  $W$ .

2. If  $s_{n-1}$  does belong to  $T$ , then  $T - \{s_{n-1}\}$  is a solution to the Knapsack problem with items  $\{s_0, s_1, \dots, s_{n-2}\}$  weights  $\{w_0, w_1, \dots, w_{n-2}\}$  values  $\{v_0, v_1, \dots, v_{n-2}\}$  and max weight  $W - w_{n-1}$ .

This shows that  $T$  is built up from solutions to subproblems, and suggests a recursive solution that is similar to the recursive solution to SubsetSum.

# Optional: Knapsack “Bottom Up” Solution

**Knapsack Optimization Problem** Given a set  $S = \{s_0, s_1, \dots, s_{n-1}\}$  of  $n$  items with positive integer weights given by  $w[] = \{w_0, w_1, \dots, w_{n-1}\}$  and nonnegative integer values  $v[] = \{v_0, v_1, \dots, v_{n-1}\}$  and a maximum weight  $W$  (a positive integer), find  $T \subseteq S$  so that  $\sum_{s_i \in T} v_i$  is maximal (the *maximum benefit*), subject to the constraint that  $\sum_{s_i \in T} w_i \leq W$ .

We describe the “bottom-up” solution, obtained by building the memoization table recursively (as was done for SubsetSum). For  $0 \leq i \leq n - 1$  and  $0 \leq j \leq W$ , we obtain an  $n \times (W + 1)$  matrix  $A$  of subsets of  $S$ .

$$A[i, j] = T \subseteq S \text{ where } \sum_{s_r \in T} w_r \leq j \text{ and } \sum_{s_r \in T} v_r \text{ is maximal.}$$

*Recursive procedure to populate the matrix A.*

**Row 0:**

$$A[0, t] = \begin{cases} \emptyset & \text{if } t < w_0 \\ \{s_0\} & \text{if } t \geq w_0 \end{cases}$$

**Row  $i$ :**

$$T_a = A[i - 1, j], T_b = A[i - 1, j - w_i] \cup \{s_i\}, B_a = \sum_{s_r \in T_a} v_r, B_b = \sum_{s_r \in T_b} v_r.$$

$$A[i, j] = \begin{cases} T_a & \text{if } B_a \geq B_b \\ T_b & \text{otherwise.} \end{cases}$$

# Optional: Knapsack “Bottom Up” Solution

- ◆ See the Demo  
DynamicKnapsack-Demo.pdf
- ◆ The set stored in  $A[n-1, W]$  is the solution.
- ◆ Running time is  $O(nW)$  in terms of values, but, in terms of input size, it's  $O(n * 2^{\text{length}(W)})$ , which is potentially exponential in  $n$  (whenever  $n \leq W$ ). Therefore, as in the case of SubsetSum, this algorithm for Knapsack runs in *pseudo-polynomial time*.

# Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Dynamic programming can transform an infeasible (exponential) computation into one that can be done efficiently.
2. Dynamic programming is applicable when many subproblems of a recursive algorithm overlap and have to be repeatedly computed. The algorithm stores solutions to subproblems so they can be retrieved later rather than having to re-compute them.
3. Transcendental Consciousness is the silent, unbounded home of all the laws of nature.
4. *Impulses within Transcendental Consciousness:* The dynamic natural laws within this unbounded field are perfectly efficient when governing the activities of the universe.
5. *Wholeness moving within itself:* In Unity Consciousness, one experiences the laws of nature and all activities of the universe as waves of one's own unbounded pure consciousness.